COS 226 Lecture 7: Searching overview

Abstract Data Types Revisited

Symbol Table, Dictionary
- items with keys
- INSERT
- SEARCH

Searching = "implement a symbol table ADT"

ADT orientation integral to modern languages
implementation in C possible
Refs: Sedgewick section 4.8
- Hanson, "C Interfaces and Implementations"

COS 217:
- ADTs control complexity in large systems

COS 226:
- ADTs precisely define implementation

Both:
- ADTs help us build layers of abstraction

Items and keys

ADT for use by clients and implementations
of symbol table ADTs
Sample INTERFACE for char items and keys

```
.  typedef char Item;
.  typedef char Key;
.  #define key(A) A
.  #define less(A, B)  (A < B)
.  #define eq(A, B)  ((A) == (B))
.  #define maxKey 256
.  #define NULLitem 0

.  Item ITEMrand(void);
.   int ITEMscan(Item *);
.  void ITEMshow(Item);
```

CLIENTS
use functions to create and write keys
manipulate items
Search algorithms (ST IMPLEMENTATION)
store and retrieve items
use "key", "less" to extract, compare keys

"key" and "less" could be in implementation

GOAL: implementations independent of item type

Symbol table ADT

Items with keys

Two basic operations

* insert
* search (find item with given key)

generic operations common to many ADTs

* create
* test if empty
* destroy (often ignored if not harmful)

other operations

* construct (batch inserts)
* sort (often a byproduct)
* delete
* find kth largest
* join

(ST > PQ)

INTERFACE for basic and generic operations

```
.       void STinit(int);
.       void STinsert(Item);
.       Item STsearch(Key);
.        int STcount();
```

Could combine with datatype interface

Should also specify

constraints and error conditions

Typical brute-force implementation

Use linear search in an ordered array

```
static Item *st;
static int N;
void STinit(int maxN)
   { st = malloc((maxN)*sizeof(Item)); N=0; }
int STcount()
   { return N; }
void STinsert(Item item)
   { int j = N++; Key v = key(item);
      while (j>0 && less(v, key(st[j-1])))
         {  st[j] = st[j-1]; j--; }
      st[j] = item;
   }
Item STsearch(Key v)
   { int j;
      for (j = 0; j < N; j++)
         { if (eq(v, key(st[j]))) return st[j];
            if (less(v, key(st[j]))) break; }
      return NULLitem;
   }
```

Other options for brute-force implementations

* unordered array
* unordered singly-linked list
* ordered singly-linked list

Binary search

Maintain ordered array

Do search in lgN steps by divide-and-conquer

```
Item search(int l, int r, Key v)
   { int m = (l+r)/2;
      if (l > r) return NULLitem;
      if eq(v, key(st[m])) return st[m];
      if (l == r) return NULLitem;
      if less(v, key(st[m]))
           return search(l, m-1, v);
      else return search(m+1, r, v);
   }
Item STsearch(Key v)
   { return search(0, N-1, v); }
```

Ex: search for I

```
.   0  1  2  3  4  5  6  7  8  9 10 11 12
.      A  A  E  G  I  M  N  O  R  S  T  X
.            l     r    m
.            1    12    6
.            1     5    3
.            4     5    4
.            5     5
```

Worst-case number of steps for binary search =
number of bits in binary representation of N

Proof: they satisfy the same recurrence

Interpolation search

Maintain ordered array

Improve upon binary search by using key values
to guess where to look

Ex: search for I
(I−A)/(X−A) = (9−1)/(24−1) = .35
therefore divide at .35, not .5
next probe will be on or close

```
.   0  1  2  3  4  5  6  7  8  9 10 11 12
.      A  A  E  G  I  M  N  O  R  S  T  X

.            l     r    m
.            1    12    5
```

THM: Interpolation search takes lglgN steps
for random files

Proof: difficult

Less than 5 probes in practice (2^2^5 = billions)

Caveats:
files not random
calculations cost more than comparisons

ST implementations cost summary

Per-operation time costs as function of ST size

	worst-case		average-case	
	insert	search	insert	search
ordered				
array	N	N	N/2	N/2
list	N	N	N/2	N/2
unordered				
array	1	N	1	N/2
list	1	N	1	N/2
binary search	N	lgN	N/2	lgN
BST	N	N	lgN	lgN
red-black	lgN	lgN	lgN	lgN
randomized	(N)	(N)	lgN	lgN
hashing	N	N	1	1

Situation further complicated by

* need to have many other operations
* operations mix
* space costs

Another algorithm design success story:

* nearly optimal worst-case cost
* fast average-case cost
* simple (but ingenious!) algorithms

First-class symbol-table ADT

Basic operations

- initialize
- insert
- search
- select kth smallest
- delete
- join two STs
- test if empty
- visit items in order

Handles (pointers to unspecified structs)

for delete, client needs to handle recs

for join, client needs to handle STs

INTERFACE for basic operations

```
.  typedef struct st* ST;
.  typedef struct STnode* STlink;
.       ST STinit(int);
.      int STcount(int);
.   STlink STinsert(ST, Item);
.     Item STsearch(ST, Key);
.     Item STselect(ST, int);
.     void STdelete(ST, STlink);
.       ST STjoin(ST, ST);
```

could have STsearch and STselect return handles

First-class ADT implementation

Concrete implementation of abstract ADT

specify data representation for records

[STlink is pointer to record (STnode)]

specify data representation for table

[ST is pointer to symbol table (st)]

Ex: Binary search trees

```
struct STnode
  { Item item; struct STlink l, r; };
struct st
  { STlink head; }
```

"What is a record in a BST?"

a node

"What is a symbol table?"

a struct with a pointer to a node (root)

Handles

record: pointer to node

symbol table:

pointer to struct with pointer to node

Providing true "abstract" data type challenging

* precise ADT specification worth the trouble
* new implementations are "plug-ins"
* compiler type-checking can be helpful
* not so challenging in some modern languages

More details: Sedgewick, sections 4.8, 9.5

Interface for standalone ADT

Use if only one ADT in application (typical)

```
.      void STinit(int);
.       int STcount();
.      void STinsert(Item);
.      Item STsearch(Key);
.      void STdelete(Item);
.      Item STselect(int);
.      void STsort(void (*visit)(Item));
```

* elimate ST handles in args
* can specify DELETE in terms of item types
* can specify JOIN in terms of ST types

Stylized code setup gives true ADT
(built-in support in C++, Java ...)

Standalone ADT

useful in most applications

allows us to focus on algorithms

Symbol table ADT client example

DEDUP a file of integers (remove duplicates)

```
#include <stdio.h>
#include <stdlib.h>
#include "Item.h"
#include "ST.h"
void main()
 {
     int v; Item item;
     STinit(maxN);
     while (scanf("%d", &v) == 1)
        {
           if (STsearch(v) != NULLitem) continue;
           key(item) = v;
           STinsert(item);
           printf("%d ", v);
        }
 }
```

Goal: independence from implementation

Space costs: same options as for PQs

- copy of data for implementation
- implementation manipulates data via indices
- implementation manipulates pointers

Examples of other clients:

- online dictionary
- compiler symbol table

BST data types

NODES consist of

- item with key
- left link to BST for smaller keys
- right link to BST for larger keys
- count of number of nodes in BST (optional)

```
typedef struct STnode* link;
struct STnode
  { Item item; link l, r; int N };
static link head, z;
link NEW(Item item, link l, link r, int N)
  { link x = malloc(sizeof *x);
    x->item = item;
    x->l = l; x->r = r; x->N = N;
    return x;
  }
void STinit()
  { head = (z = NEW(NULLitem, 0, 0, 0)); }
int STcount() { return head->N; }
```

Recursive BST implementation

Search and insertion code directly follows from BST definition

```
Item searchR(link h, Key v)
  { Key t = key(h->item);
    if (h == z) return NULLitem;
    if eq(v, t) return h->item;
    if less(v, t)
        return searchR(h->l, v);
    else return searchR(h->r, v);
  }
Item STsearch(Key v)
  { return searchR(head, v); }
link insertR(link h, Item item)
  { Key v = key(item), t = key(h->item);
    if (h == z) return NEW(item, z, z, 1);
    if less(v, t)
        h->l = insertR(h->l, item);
    else h->r = insertR(h->r, item);
    (h->N)++; return h;
  }
void STinsert(Item item)
  { head = insertR(head, item); }
```

Nonrecursive BST implementation

Travel down the tree

move left if smaller than search key

move right if larger than search key

```
STinsert(Item item)
  { Key v = key(item);
    link p = head, x = p;
    if (head == z)
      { head = NEW(item, z, z); return; }
    while (x != z)
      { p = x;
        if less(v, key(x->item))
            x = x->l;
        else x = x->r;
      }
    x = NEW(item, z, z, 1);
    if less(v, key(p->item))
        p->l = x;
    else p->r = x;
  }
```

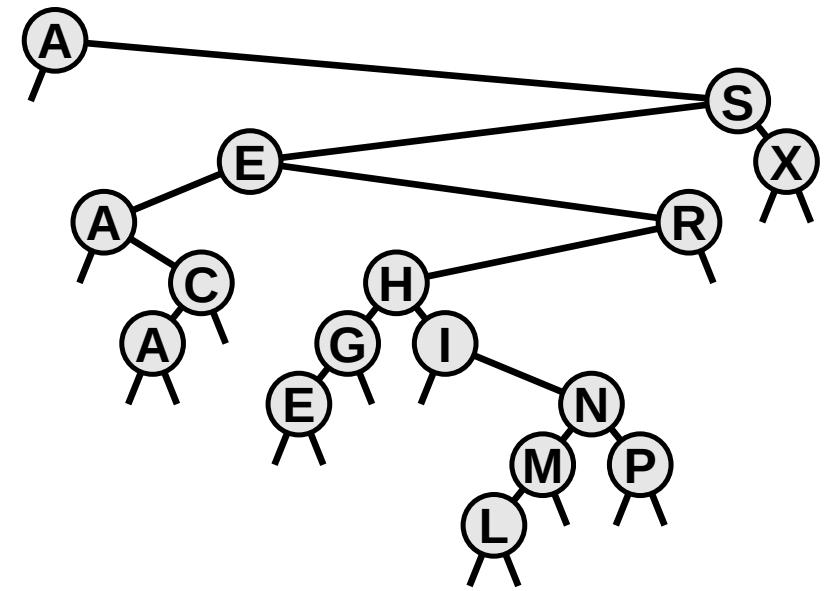
Equivalent to recursive version

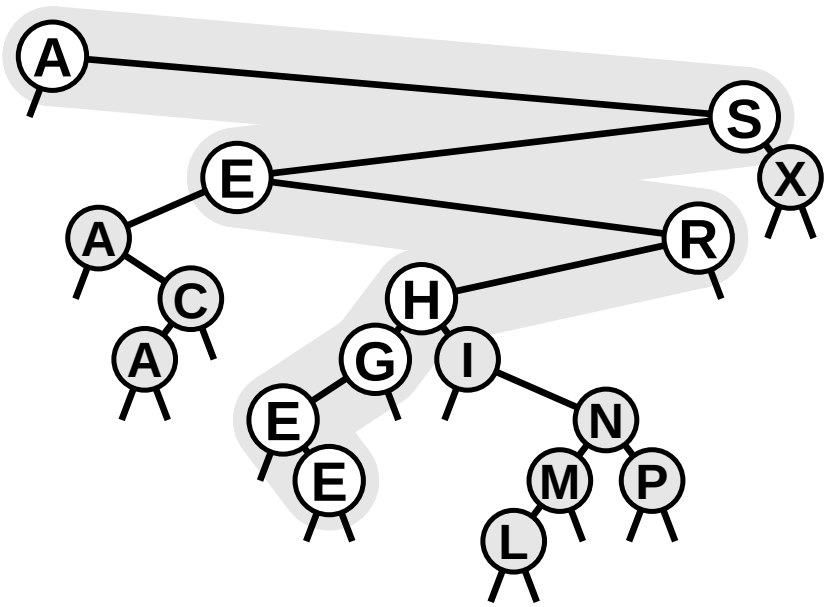
(can derive automatically)

Recursive versions of tree algs

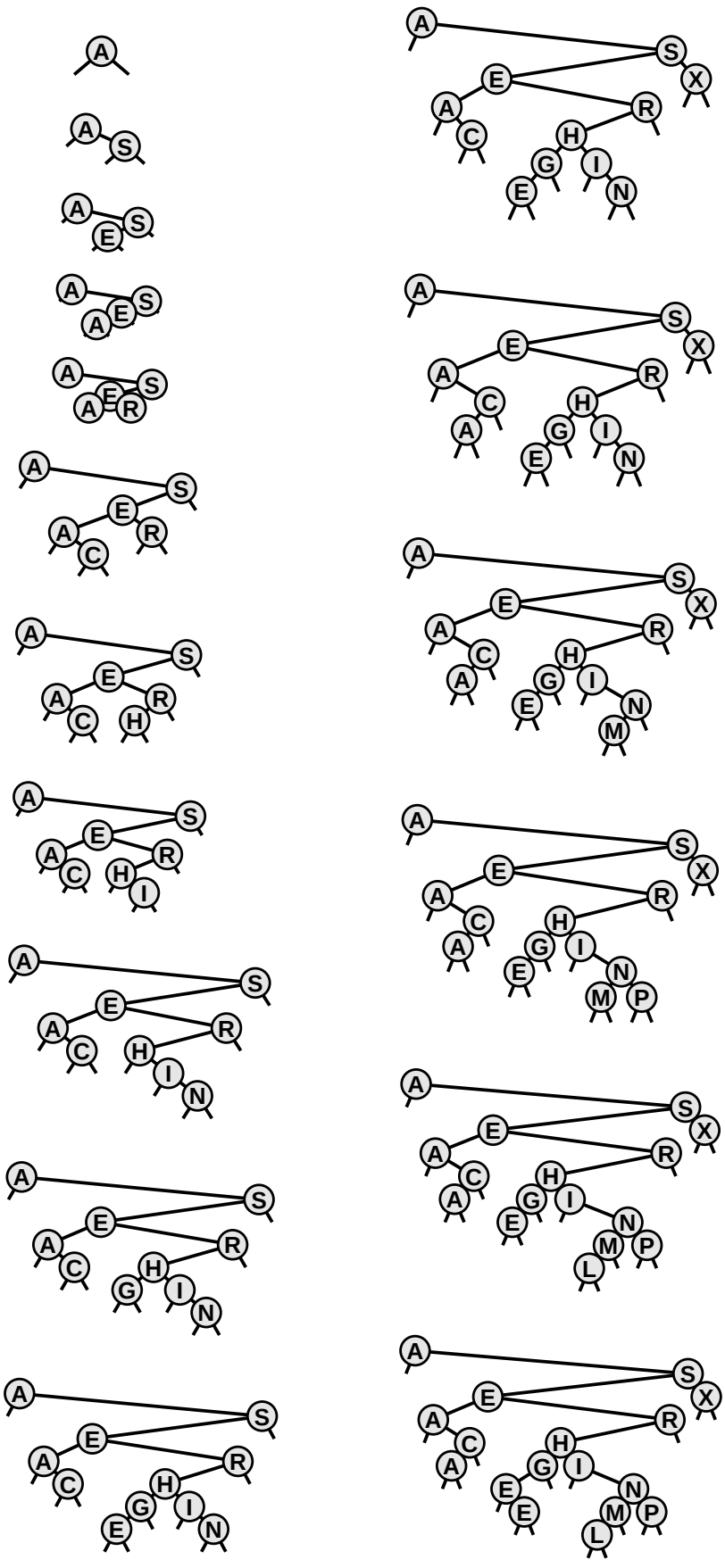
simpler to implement and understand

BST insertion example





BST construction



Cost of search in BSTs

One-to-one correspondence
between BST and Quicksort partitioning

Total cost of searching for all nodes
satisfies same recurrence as Quicksort

$$C(N) = N+1 + 2(C(1) + C(2) + \ldots + C(N-1))/N$$
$$\ldots$$
$$= 2N \ln N$$

THM: Search and insertion in BSTs involves examining 2lnN nodes on the average.

Worst case: nodes in order
degenerates to linked list, insertion cost N

Other operations in BSTs

SORT: traverse tree in inorder

```
void sortR(link h, void (*visit)(Item))
   {
      if (h == z) return;
      sortR(h->l, visit);
      visit(h->item);
      sortR(h->r, visit);
   }
void STsort(void (*visit)(Item))
   { sortR(head, visit); }
```

same cost as Quicksort (+ space for links)

Useful as "free" ST operation

GENERALIZED PQ: add tree size to nodes

```
Item selectR(link h, int k)
   { int t = h->l->N;
      if (h == z) return NULLitem;
      if (t > k ) return selectR(h->l, k);
      if (t < k ) return selectR(h->r, k-t-1);
      return h->item;
   }
Item STselect(int k)
   { return selectR(head, k); }
```

special case: go left for smallest

Delete arbitrary node? Join?

Deletion in BSTs

To delete a node at the bottom (A E L P X)

remove it

To delete a node with one child (A A C G I M)

pass the child up

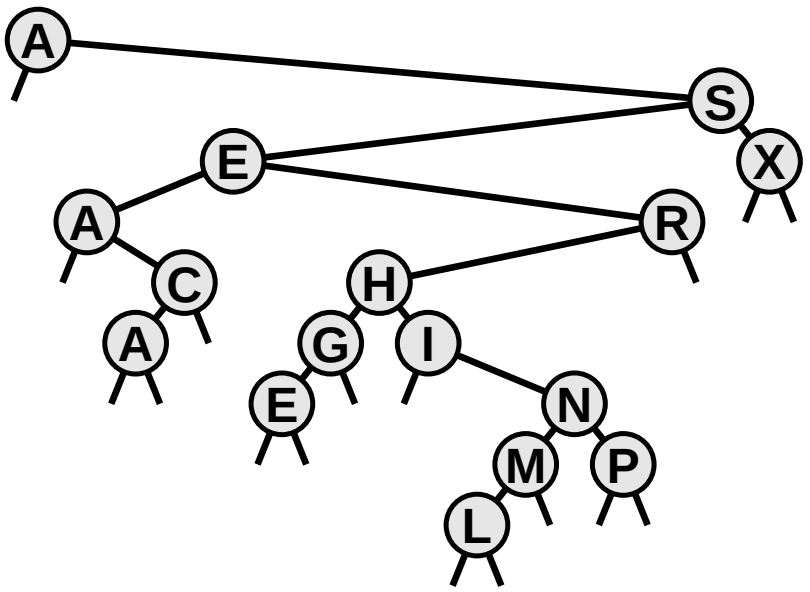
To delete a node with two children (S E H N)

find the "next" node

use right-left*

OR left-right*

swap, then remove (reduces to A or B)



Problem:

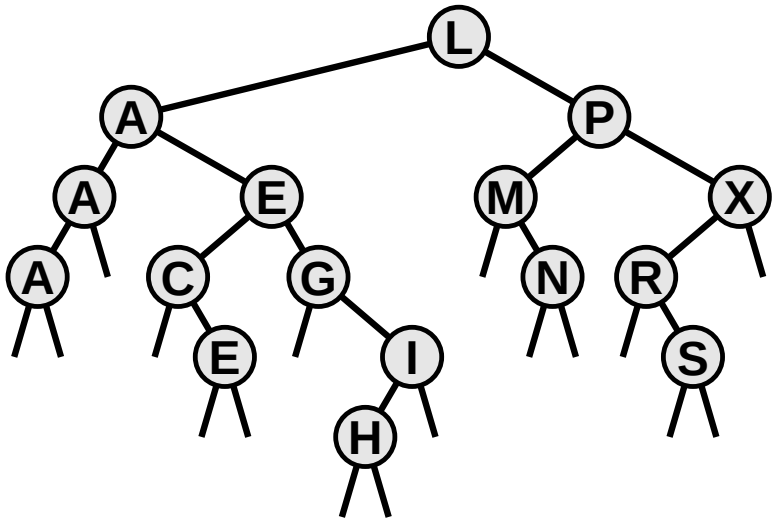
strategy not symmetric

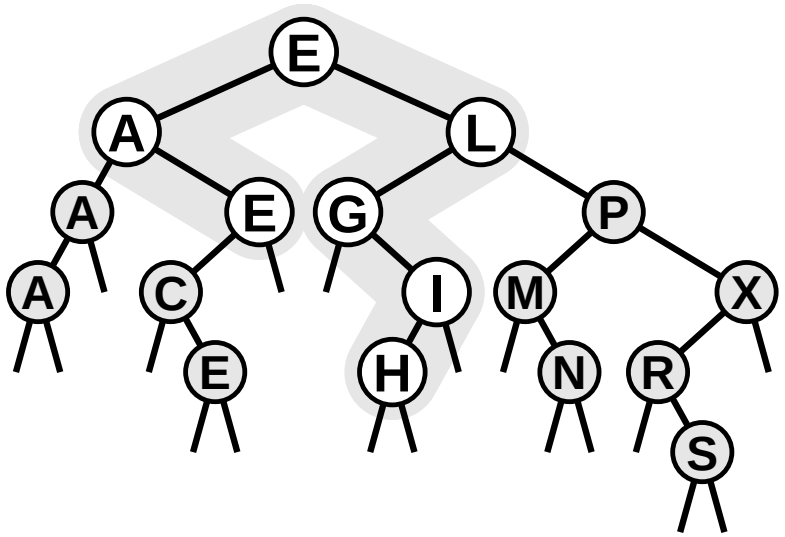
clumsy algorithm (see code in book!)

Serious problem: trees not random after long sequence of insertions and deletions (!!)

BSTs by root insertion

Idea: Insert such that new node stays at root





Nonrecursive implementation possible

pass link that would cross to parent

Recursive implementation easy

based on ROTATION

Faster search for recently inserted nodes

Nonrecursive BST root insertion

```
void BSTinsert(Item item)
  { Key v = key(item); int lr;
    struct BSTnode **t, **u, *x;
    x = NEW(rec, z, z);
    if (head == z) return x;
    if (less(v, key(head->item)))
      { x->r = head; head = x; t = &x->l;
                          x = x->r; lr = 0; }
    else
      { x->l = head; head = x; t = &x->r;
                          x = x->l; lr = 1; }
    while (x != z)
      if (less(v, key(x->item)))
      { if ( lr) { lr = !lr; *t = *u; t = u; }
        u = &x->l; x = *u; }
      else
      { if (!lr) { lr = !lr; *t = *u; t = u; }
        u = &x->r; x = *u; }
    *t = z;
    return head;
  }
```

Challenge for the bored:

check that this code is correct

Conclusion for the rest of us:

recursive implementations *are* simpler!

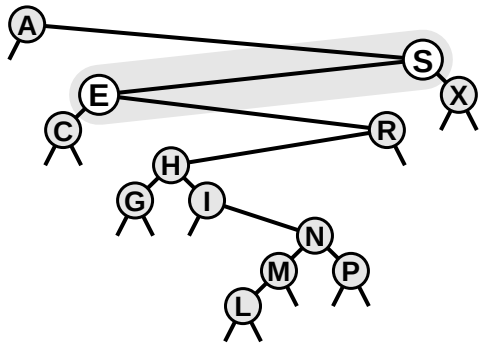
ROTATE operation in trees

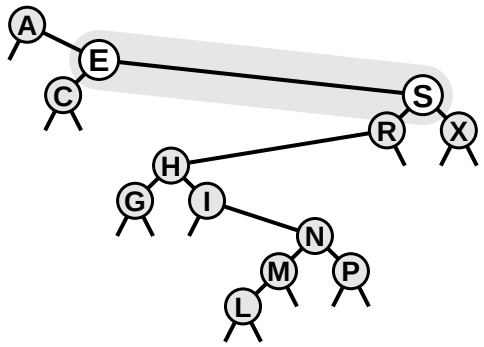
Fundamental operation

- rearrange nodes in trees
- local transformation (two links)
- maintain sorted order

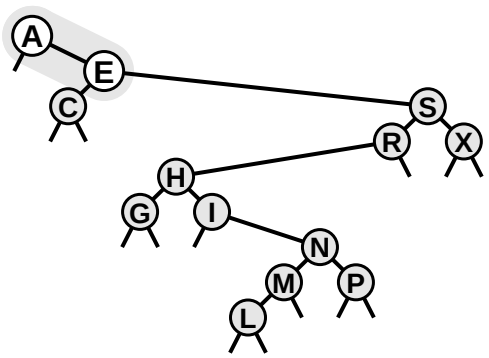
```
link rotR(link h)
   { link x = h->l; h->l = x->r; x->r = h;
     return x; }
link rotL(link h)
   { link x = h->r; h->r = x->l; x->l = h;
     return x; }
```

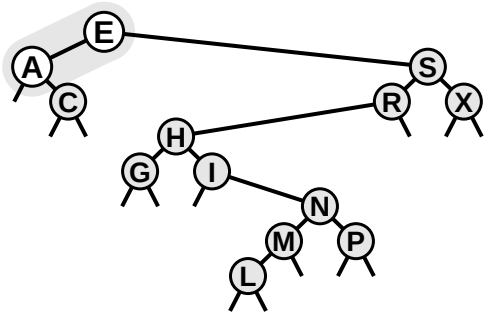
Right rotate





Left rotate





Recursive BST root insertion code

To insert at root

insert at root of subtree (recursively)

rotate to bring inserted key to root

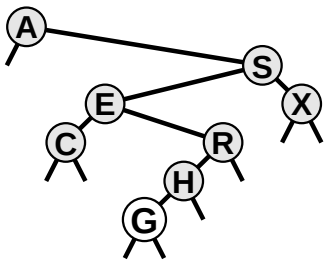
```
link insertT(link h, Item item)
  { Key v = key(item);
    if (h == z) return NEW(item, z, z, 1);
    if (less(v, key(h->item)))
      { h->l = insertT(h->l, item);
        h = rotR(h); }
    else
      { h->r = insertT(h->r, item);
        h = rotL(h); }
    return h;
  }
void STinsert(Item item)
  { head = insertT(head, item); }
```

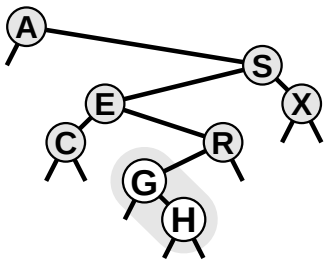
Equivalent to

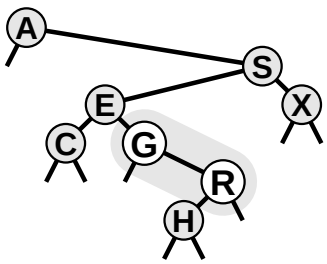
insert the node

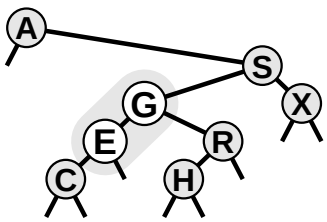
use rotations to bring it up to the top

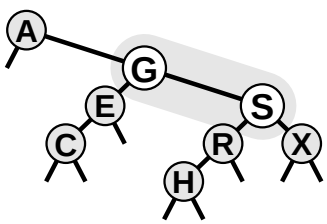
Rotate to the root

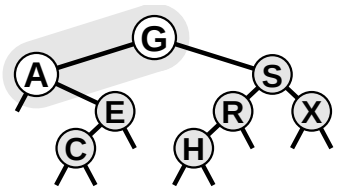




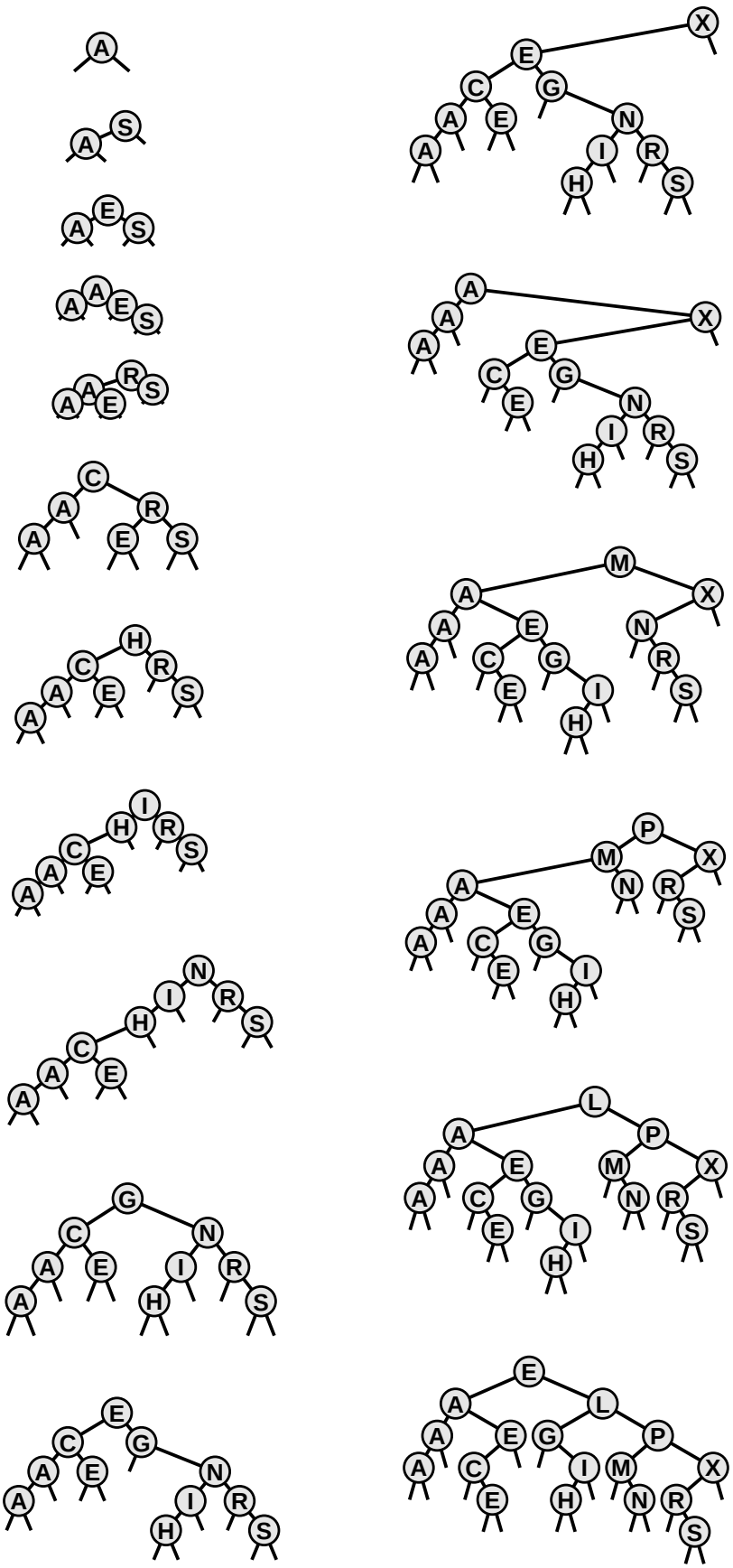








Top-down BST construction (root insertion)



Searching (to be continued)

SUMMARY

- use elementary methods for small cases
- binary search guarantees lgN steps for search
  - (but requires N steps for insertion)
- interpolation search can speed up search
  - (but requires N steps for insertion)
- binary search trees give fast
  - search and insert, average-case

Binary search trees can do more

- * insert
- * search
- * delete
- * select kth smallest
- * ...

all in O(lgN) steps, average-case
with O(NlgN) sort as a byproduct

Goal: Symbol table implementation

- with O(lgN) GUARANTEED performance
- for both search and insert